System Methodology
Holistic Performance Analysis on Modern Systems

Brendan Gregg
Senior Performance Architect
Background
History

• System Performance Analysis up to the '90s:
  – Closed source UNIXes and applications
  – Vendor-created metrics and performance tools
  – Users interpret given metrics

• Problems
  – Vendors may not provide the best metrics
  – Often had to \textit{infer}, rather than \textit{measure}
  – Given metrics, what do we do with them?

```bash
# ps alx
 F S UID   PID  PPID  CPU  PRI  NICE ADDR  SZ  WCHAN TTY TIME CMD
3:S   0     0     0   0   0   20  2253   2   4412 ?  186:14 swapper
1:S   0     1     0   0  30   20  2423   8  46520 ?   0:00 /etc/init
1:S   0    16     1   0  30   20  2273  11  46554 co  0:00 –sh
[...]
```
Today

1. Open source
   - Operating systems: Linux, BSDs, illumos, etc.
   - Applications: source online (Github)

2. Custom metrics
   - Can patch the open source, or,
   - Use dynamic tracing (open source helps)

3. Methodologies
   - Start with the questions, then make metrics to answer them
   - Methodologies can pose the questions

Biggest problem with dynamic tracing has been what to do with it. Methodologies guide your usage.
Crystal Ball Thinking
Anti-Methodologies
Street Light *Anti*-Method

1. Pick observability tools that are
   - Familiar
   - Found on the Internet
   - Found at random
2. Run tools
3. Look for obvious issues
Drunk Man *Anti*-Method

- Drink Tune things at random until the problem goes away
Blame Someone Else *Anti*-Method

1. Find a system or environment component you are not responsible for
2. Hypothesize that the issue is with that component
3. Redirect the issue to the responsible team
4. When proven wrong, go to 1
Traffic Light Anti-Method

1. Turn all metrics into traffic lights
2. Open dashboard

• Type I errors: red instead of green
  – team wastes time
• Type II errors: green instead of red
  – performance issues undiagnosed
  – team wastes more time looking elsewhere

Traffic lights are suitable for *objective* metrics (eg, errors), not *subjective* metrics (eg, IOPS, latency).
Methodologies
Performance Methodologies

- For system engineers:
  - ways to analyze unfamiliar systems and applications
- For app developers:
  - guidance for metric and dashboard design

System Methodologies:
- Problem statement method
- Functional diagram method
- Workload analysis
- Workload characterization
- Resource analysis
- USE method
- Thread State Analysis
- On-CPU analysis
- CPU flame graph analysis
- Off-CPU analysis
- Latency correlations
- Checklists
- Static performance tuning
- Tools-based methods

Collect your own toolbox of methodologies
Problem Statement Method

1. What makes you **think** there is a performance problem?
2. Has this system **ever** performed well?
3. What has **changed** recently?
   - software? hardware? load?
4. Can the problem be described in terms of **latency**?
   - or run time. not IOPS or throughput.
5. Does the problem affect **other** people or applications?
6. What is the **environment**?
   - software, hardware, instance types? versions? config?
Functional Diagram Method

1. Draw the functional diagram
2. Trace all components in the data path
3. For each component, check performance

Breaks up a bigger problem into smaller, relevant parts

Eg, imagine throughput between the UCSB 360 and the UTAH PDP10 was slow…

ARPA Network 1969
Workload Analysis

• Begin with application metrics & context
• A **drill-down** methodology

• Pros:
  – Proportional, accurate metrics
  – App context

• Cons:
  – App specific
  – Difficult to dig from app to resource
Workload Characterization

• Check the workload: **who, why, what, how**
  – not resulting performance

• Eg, for CPUs:
  1. Who: which PIDs, programs, users
  2. Why: code paths, context
  3. What: CPU instructions, cycles
  4. How: changing over time
Workload Characterization: CPUs

**Who**
- `top` command

**Why**
- CPU sample
- Flame graphs

**How**
- Monitoring

**What**
- PMCs
Resource Analysis

• Typical approach for system performance analysis: begin with system tools & metrics
  
• Pros:
  – Generic
  – Aids resource perf tuning

• Cons:
  – Uneven coverage
  – False positives

![Diagram showing the layers of system analysis from Workload to Hardware]
The USE Method

• For every resource, check:
  1. **Utilization**: busy time
  2. **Saturation**: queue length or time
  3. **Errors**: easy to interpret (objective)

Starts with the questions, then finds the tools
Eg, for hardware, check every resource incl. busses:

![Diagram of hardware components]

USE Method: Rosetta Stone of Performance Checklists

The following USE Method example checklists are automatically generated from the individual pages for: Linux, Solaris, Mac OS X, and FreeBSD. These analyze the performance of the physical host. You can customize this table using the checkboxes on the right.

There are some additional USE Method example checklists not included in this table: the SmartOS checklist, which is for use within an OS virtualized guest, and the Unix 7th Edition checklist for historical interest.

For general purpose operating system differences, see the Rosetta Stone for Unix, which was the inspiration for this page.

### Hardware Resources


<table>
<thead>
<tr>
<th>Resource</th>
<th>Metric</th>
<th>Linux</th>
<th>Solaris</th>
<th>FreeBSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>errors</td>
<td>perf (LPE) if processor specific error events (CPC) are available; eg, AMD64's &quot;04Ah Single-bit ECC Errors Recorded by Scrubber&quot; [4]</td>
<td>fmadm faulty; cpustat (CPC) for whatever error counters are supported (eg, thermal throttling)</td>
<td>dmesg; /var/log/messages; pmcstat for PMC and whatever error counters are supported (eg, thermal throttling)</td>
</tr>
<tr>
<td>CPU</td>
<td>saturation</td>
<td>system-wide: vmstat 1, &quot;r&quot; &gt; CPU count [2]; sar -q, &quot;runq-sz&quot; &gt; CPU count; dstat -p, &quot;run&quot; &gt; CPU count; per-process: /proc/PID/schedstat 2nd field (sched_info.run_delay); perf sched latency (shows &quot;Average&quot; and &quot;Maximum&quot; delay per-schedule); dynamic tracing, eg, SystemTap schedtimes.stp &quot;queued(us)&quot; [3]</td>
<td>system-wide: uptime, load averages; vmstat 1, &quot;r&quot;; DTrace dispqlen.d (DTI) for a better &quot;vmstat r&quot;; per-process: prstat -mLc 1, &quot;LAT&quot;</td>
<td>system-wide: uptime, &quot;load averages&quot; &gt; CPU count; vmstat 1, &quot;procs:&quot; &gt; CPU count; per-cpu: DTrace to profile CPU run queue lengths [1]; per-process: DTrace of scheduler events [2]</td>
</tr>
<tr>
<td>CPU</td>
<td>utilization</td>
<td>system-wide: vmstat 1, &quot;us&quot; + &quot;sy&quot; + &quot;st&quot;; sar -u, sum fields except &quot;%idle&quot; and &quot;%iowait&quot;; dstat -c, sum fields except &quot;idl&quot; and &quot;wai&quot;; per-cpu: mpstat -P ALL 1, sum fields except &quot;%idle&quot; and &quot;%iowait&quot;; sar -P ALL, same as mpstat; per-process: top, &quot;%CPU&quot;; htop, &quot;CPU%&quot;; ps -o pcpu; pidstat 1, &quot;%CPU&quot;; per-kernel-thread: top / htop (&quot;K&quot; to toggle), where VIRT == 0 (heuristic). [1]</td>
<td>per-cpu: mpstat 1, &quot;usr&quot; + &quot;sys&quot;; system-wide: vmstat 1, &quot;us&quot; + &quot;sy&quot;; per-process: prstat -c 1 (&quot;CPU&quot; == recent); prstat -mLc 1 (&quot;USR&quot; + &quot;SYS&quot;); per-kernel-thread: lockstat -i&quot; rate, DTrace profile stack()</td>
<td>system-wide: vmstat 1, &quot;us&quot; + &quot;sy&quot;; per-cpu: vmstat -P; per-process: top, &quot;WCPU&quot; for weighted and recent usage; per-kernel-process: top -S, &quot;WCPU&quot;</td>
</tr>
<tr>
<td>CPU</td>
<td>intercon</td>
<td>LPE (CPC) for whatever is available</td>
<td>cpustat (CPC) for whatever is available</td>
<td>pmcstat and relevant PMCs for whatever is available</td>
</tr>
<tr>
<td>CPU</td>
<td>saturation</td>
<td>LPE (CPC) for stall modes</td>
<td>cpustat (CPC) for stall</td>
<td>pmcstat and relevant PMCs for stall</td>
</tr>
</tbody>
</table>
USE Method: Unix 7th Edition Performance Checklist

Out of curiosity, I've developed a USE Method-based performance checklist for Unix 7th Edition on a PDP-11/45, which I've been running via a PDP simulator. 7th Edition is from 1979, and was the first Unix with iostat(1M) and pstat(1M), enabling more serious performance analysis from shipped tools. Were I to write a checklist for earlier Unixes, it would contain many more "unknowns".

I often work on the illumos kernel, a direct descendant of Unix which contains some original AT&T code. It's been interesting to study this earlier version, and see familiar code that has survived over 30 years of development.

Example screenshots from various tools are shown at the end of this page.

Physical Resources

<table>
<thead>
<tr>
<th>component</th>
<th>type</th>
<th>metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>utilization</td>
<td>system-wide: iostat 1, utilization is &quot;user&quot; + &quot;nice&quot; + &quot;sysm&quot;; per-process: ps alx, &quot;CPU&quot; shows recent CPU usage (max 255), and &quot;TIME&quot; shows cumulative minutes:seconds of CPU time</td>
</tr>
<tr>
<td>CPU</td>
<td>saturation</td>
<td>ps alx</td>
</tr>
<tr>
<td>CPU</td>
<td>errors</td>
<td>console message if lucky, otherwise panic</td>
</tr>
<tr>
<td>Memory capacity</td>
<td>utilization</td>
<td>system-wide: unknown [1]; per-type: unknown [2]; per-process: ps alx, &quot;SZ&quot; is the in-core (main memory) in blocks (512 bytes); pstat -p, &quot;SIZE&quot; is in-core size, in units of core clicks (64 bytes) and printed in octal!</td>
</tr>
<tr>
<td>Memory capacity</td>
<td>saturation</td>
<td>system-wide: iostat 1, sustained &quot;tpm&quot; may be caused by swapping to disk; significant delays as processes wait for space to swap in</td>
</tr>
<tr>
<td>Memory capacity</td>
<td>errors</td>
<td>malloc() returns 0; ENOMEM</td>
</tr>
<tr>
<td>Disk I/O</td>
<td>utilization</td>
<td>system-wide: iostat -i 1, &quot;IO active&quot; plus &quot;IO wait&quot; percents; per-disk-controller: iostat -i 1, RF, RK, RP &quot;active&quot; percents; rough estimate using iostat 1, and &quot;tpm&quot; for transactions per minute on expected max; per-disk: listen to each rattle; unknown from Unix, unless only 1 disk per controller; per-process: unknown</td>
</tr>
<tr>
<td>Disk I/O</td>
<td>saturation</td>
<td>unknown [3]</td>
</tr>
<tr>
<td>Disk I/O</td>
<td>errors</td>
<td>might get a console message, eg, &quot;err on dev&quot;, &quot;ECC on dev&quot; or &quot;no space on dev&quot;, otherwise unknown [4]</td>
</tr>
<tr>
<td>Tape I/O</td>
<td>utilization</td>
<td>look at tape drives and watch them spin [5]</td>
</tr>
</tbody>
</table>
Figure 3-2.4. Primary Guidance Path - Simplified Block Diagram
USE Method: Software

• USE method can also work for software resources
  – kernel or app internals, cloud environments
  – small scale (eg, locks) to large scale (apps). Eg:

• Mutex locks:
  – utilization → lock hold time
  – saturation → lock contention
  – errors → any errors

• Entire application:
  – utilization → percentage of worker threads busy
  – saturation → length of queued work
  – errors → request errors
RED Method

- For every service, check that:
  1. **Request** rate
  2. **Error** rate
  3. **Duration** (distribution)

are within SLO/A

Another exercise in posing questions from functional diagrams

By Tom Wilkie: http://www.slideshare.net/weaveworks/monitoring-microservices
Thread State Analysis

Identify & quantify time in states
Narrows further analysis to state
Thread states are applicable to all apps
### TSA: eg, Solaris

#### 1) `$ prstat -mLc 1`

<table>
<thead>
<tr>
<th>PID</th>
<th>USERNAME</th>
<th>USR</th>
<th>SYS</th>
<th>TRP</th>
<th>TFL</th>
<th>DFL</th>
<th>LCK</th>
<th>SLP</th>
<th>LAT</th>
<th>VCX</th>
<th>ICX</th>
<th>SCL</th>
<th>SIG</th>
<th>PROCESS/LWPID</th>
</tr>
</thead>
<tbody>
<tr>
<td>45747</td>
<td>1000</td>
<td>35</td>
<td>28</td>
<td>0.0</td>
<td>0.0</td>
<td>22</td>
<td>0.0</td>
<td>16</td>
<td>0.1</td>
<td>216</td>
<td>93</td>
<td>38K</td>
<td>0</td>
<td>beam.smp/192</td>
</tr>
</tbody>
</table>

[...]

#### 2)

<table>
<thead>
<tr>
<th>Fields</th>
<th>States</th>
<th>Analysis</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>USR+SYS</td>
<td>Executing</td>
<td>1. Profile stacks using DTrace; Flame Graphs</td>
<td>Look for inefficiencies</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. Check CPU stall cycles: cpustat, DTrace</td>
<td>Look for tunables/config in active code</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. If SYS time, analyze syscalls using DTrace</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. Check system-wide memory free: vmstat 1</td>
<td>Increase memory limits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. Check any resource controls; eg: zonememstat</td>
<td>Look for leaks/growth</td>
</tr>
<tr>
<td>LCK</td>
<td>Lock + Idle</td>
<td>1. Coarse: profile CPU stacks and look for spins</td>
<td>Check config</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. Analyze using DTrace [p]lockstat providers</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. Separate locks and the Idle state using DTrace</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sched::off-cpu with ustack()</td>
<td></td>
</tr>
<tr>
<td>SLP</td>
<td>Sleep + Idle</td>
<td>1. Quick resource check: iostat -xnz 1, nicstat 1</td>
<td>Tune or upgrade resource</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. Identify both sleep reason and separate from Idle:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DTrace sched::off-cpu with ustack() and stack()</td>
<td></td>
</tr>
<tr>
<td>LAT</td>
<td>Runnable</td>
<td>1. Check system CPU usage: mpstat 1</td>
<td>Upgrade CPUs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. Check any resource controls; eg, prctl,</td>
<td>Increase CPU limits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>kstat -p caps::cpucaps_zone*:</td>
<td>Move/tune other load</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. Check for pbind/psets limiting migrations</td>
<td>Unbind apps</td>
</tr>
</tbody>
</table>
TSA: eg, RSTS/E

RSTS: DEC OS from the 1970's

TENEX (1969-72) also had Control-T for job states

<table>
<thead>
<tr>
<th>State Column (Job Status)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RN</td>
</tr>
<tr>
<td>RS</td>
</tr>
<tr>
<td>BF</td>
</tr>
<tr>
<td>SL</td>
</tr>
<tr>
<td>SR</td>
</tr>
<tr>
<td>FP</td>
</tr>
<tr>
<td>TT</td>
</tr>
<tr>
<td>HB</td>
</tr>
<tr>
<td>KB</td>
</tr>
<tr>
<td>^C</td>
</tr>
<tr>
<td>CR</td>
</tr>
<tr>
<td>MT, MM, or MS</td>
</tr>
<tr>
<td>LP</td>
</tr>
<tr>
<td>DT</td>
</tr>
<tr>
<td>DK, DM, DB, DP, DL, DR</td>
</tr>
</tbody>
</table>
TSA: eg, OS X

Instruments: Thread States
On-CPU Analysis

1. Split into user/kernel states
   – /proc, vmstat(1)
2. Check CPU balance
   – mpstat(1), CPU utilization heat map
3. Profile software
   – User & kernel stack sampling (as a **CPU flame graph**)
4. Profile cycles, caches, busses
   – PMCs, CPI flame graph
CPU Flame Graph Analysis

1. Take a CPU profile
2. Render it as a flame graph
3. Understand all software that is in >1% of samples

Discovers issues by their CPU usage
- Directly: CPU consumers
- Indirectly: initialization of I/O, locks, times, ...

Narrows target of study to only running code
- See: "The Flame Graph", CACM, June 2016
Java Mixed-Mode CPU Flame Graph

- eg, Linux perf_events, with:
- Java -XX:+PreserveFramePointer
- Java perf-map-agent
CPI Flame Graph

- Profile cycle stack traces and instructions or stalls separately
- Generate CPU flame graph (cycles) and color using other profile
- eg, FreeBSD: pmcstat

red == instructions
blue == stalls
Off-CPU Analysis

Analyze off-CPU time via blocking code path:

Off-CPU flame graph

Often need **wakeup** code paths as well...
Off-CPU Time Flame Graph

Trace blocking events with kernel stacks & time blocked (eg, using Linux BPF)
… can also associate wake-up stacks with off-CPU stacks (eg, Linux 4.6: samples/bpf/offwaketime*)
Associate more than one waker: the full chain of wakeups

With enough stacks, all paths lead to metal

An approach for analyzing all off-CPU issues
Latency Correlations

1. Measure latency histograms at different stack layers
2. Compare histograms to find latency origin

Even better, use latency heat maps
• Match outliers based on both latency and time
Checklists: eg, Linux Perf Analysis in 60s

1. uptime -------------------------------► load averages
2. dmesg | tail ------------------------► kernel errors
3. vmstat 1 --------------------------► overall stats by time
4. mpstat -P ALL 1 ----------------► CPU balance
5. pidstat 1 --------------------------► process usage
6. iostat -xz 1 ----------------► disk I/O
7. free -m --------------------------► memory usage
8. sar -n DEV 1 ----------------► network I/O
9. sar -n TCP,ETCP 1 ----------------► TCP stats
10. top -----------------------------► check overview

Checklists: eg, Netflix perfvitals Dashboard

1. RPS, CPU
2. Volume
3. Instances
4. Scaling
5. CPU/RPS
6. Load Avg
7. Java Heap
8. ParNew
9. Latency
10. 99th tile
Static Performance Tuning: eg, Linux

Operating System

Applications

System Libraries

System Call Interface

Software

Hardware

Various:
- sysctl /sys
dmesg lshw

/proc/cpuinfo
cpuid lscpu

CPU Interconnect

Memory Bus

I/O Bus

I/O Bridge

F/W Config

Device Drivers

Linux Kernel

App Config

df

dmesg ip route

mdadm

Megacli

VFS

Sockets

Scheduler

TCP/UDP

Clocksource

Volume Manager

IP

Virtual Memory

Block Device Interface

Ethernet

I/O Controller

Network Controller

Disk

Swap

Port

Port

lsblk lsscsi blockdev
smartctl fdisk -l

swapon

1. Try all the tools! May be an anti-pattern. Eg, OS X:
Other Methodologies

• Scientific method
• 5 Why's
• Process of elimination
• Intel's Top-Down Methodology
• Method R
What You Can Do
What you can do

1. Know what's now possible on modern systems
   – Dynamic tracing: efficiently instrument any software
   – CPU facilities: PMCs, MSRs (model specific registers)
   – Visualizations: flame graphs, latency heat maps, …

2. Ask questions first: use methodologies to ask them

3. Then find/build the metrics

4. Build or buy dashboards to support methodologies
Dynamic Tracing: Efficient Metrics

Eg, tracing TCP retransmits

Old way: packet capture

tcpdump
1. read
2. dump

Analyzer
1. read
2. process
3. print

New way: dynamic tracing

Tracer
1. configure
2. read

tcp_retransmit_skb()

Kernel

buffer
send
receive
file system
disks
Dynamic Tracing: Measure Anything

Those are Solaris/DTrace tools. Now becoming possible on all OSes: FreeBSD & OS X DTrace, Linux BPF, Windows ETW
Performance Monitoring Counters

Eg, FreeBSD PMC groups for Intel Sandy Bridge:
Visualizations

Eg, Disk I/O latency as a heat map, quantized in kernel:

```
root@bgregg-test-i-23e01ede:/mnt/src/linux-4.0.0+/samples/bpf# ./traceX3
heatmap of I/O latency
■ — many events with this latency
□ — few events

<table>
<thead>
<tr>
<th>latency</th>
<th>passage of time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1us</td>
<td>10us</td>
</tr>
<tr>
<td>100us</td>
<td>1ms</td>
</tr>
<tr>
<td>10ms</td>
<td>100ms</td>
</tr>
<tr>
<td>1s</td>
<td>10s</td>
</tr>
</tbody>
</table>
```

- high-latency outliers
- bulk of I/O
- low-latency mode
- passage of time
USE Method: eg, Netflix Vector

- CPU:
  - utilization
  - saturation

- Network:
  - utilization
  - load
  - saturation

- Memory:
  - utilization
  - saturation

- Disk:
  - load
  - utilization
  - saturation
USE Method: To Do

Showing what is and is not commonly measured
CPU Workload Characterization: To Do

Showing what is and is not commonly measured
Summary

• It is the crystal ball age of performance observability
• What matters is the questions you want answered
• Methodologies are a great way to pose questions
References & Resources

• USE Method
  – http://queue.acm.org/detail.cfm?id=2413037
• TSA Method
  – http://www.brendangregg.com/tsamethod.html
• Off-CPU Analysis
  – http://www.brendangregg.com/offcpuanalysis.html
  – http://www.brendangregg.com/blog/2016-02-05/ebpf-chaingraph-prototype.html
• Static Performance Tuning, Richard Elling, Sun blueprint, May 2000
• RED Method: http://www.slideshare.net/weaveworks/monitoring-microservices
• Other system methodologies
  – Systems Performance: Enterprise and the Cloud, Prentice Hall 2013
• Flame Graphs
  – http://queue.acm.org/detail.cfm?id=2927301
  – http://www.brendangregg.com/flamegraphs.html
• Latency Heat Maps
  – http://queue.acm.org/detail.cfm?id=1809426
• ARPA Network: http://www.computerhistory.org/internethistory/1960s
• DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD, Prentice Hall 2011
• Questions?
• http://slideshare.net/brendangregg
• http://www.brendangregg.com
• bgregg@netflix.com
• @brendangregg